|  |
| --- |
| Experiment No. 7 |
| Program for data structure using built in function for link list, stack and queues |
| Date of Performance: 4/3/2024 |
| Date of Submission:11/3/2024 |

**Experiment No. 7**

**Title:** Program for data structure using built in function for link list, stack and queues

**Aim:** To study and implement data structure using built in function for link list, stack and queues

**Objective:** To introduce data structures in python **Theory:**

Stacks -the simplest of all data structures, but also the most important. A stack is a collection of objects that are inserted and removed using the LIFO principle. LIFO stands for “Last In First Out”. Because of the way stacks are structured, the last item added is the first to be removed, and vice-versa: the first item added is the last to be removed.

Queues – essentially a modified stack. It is a collection of objects that are inserted and removed according to the FIFO (First In First Out) principle. Queues are analogous to a line at the grocery store: people are added to the line from the back, and the first in line is the first that gets checked out – BOOM, FIFO!

Linked Lists

The Stack and Queue representations I just shared with you employ the python-based list to store their elements. A python list is nothing more than a dynamic array, which has some disadvantages.

The length of the dynamic array may be longer than the number of elements it stores, taking up precious free space.

Insertion and deletion from arrays are expensive since you must move the items next to them over

Using Linked Lists to implement a stack and a queue (instead of a dynamic array) solve both of these issues; addition and removal from both of these data structures (when implemented with a linked list) can be accomplished in constant O(1) time. This is a HUGE advantage when dealing with lists of millions of items.

Linked Lists – comprised of ‘Nodes’. Each node stores a piece of data and a reference to its next and/or previous node. This builds a linear sequence of nodes. All Linked Lists store a head, which is a reference to the first node. Some Linked Lists also store a tail, a reference to the last node in the list.

**Code:**

**Stack:**

class Stack:

def \_\_init\_\_(self):

self.items = []

def is\_empty(self): return len(self.items) == 0

def push(self, item): self.items.append(item)

def pop(self):

if not self.is\_empty(): return self.items.pop() else:

return "Stack is empty"

def peek(self): if not self.is\_empty():

return self.items[-1] else:

return "Stack is empty"

def size(self):

return len(self.items) stack = Stack() while True: print("\nStack Operations:") print("1. Push") print("2. Pop") print("3. Peek") print("4. Size") print("5. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

item = input("Enter item to push: ") stack.push(item) print("Item", item, "pushed onto the stack") elif choice == 2:

popped\_item = stack.pop() print("Popped item:", popped\_item) elif choice == 3:

top\_item = stack.peek() print("Top item:", top\_item) elif choice == 4: print("Size of stack:", stack.size()) elif choice == 5:

print("Exiting...") break

else: print("Invalid choice. Please enter a valid option.")

**Output of Stack:** Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 1

Enter item to push: 7

Item 7 pushed onto the stack

Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 1

Enter item to push: 8

Item 8 pushed onto the stack

Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 1

Enter item to push: 9

Item 9 pushed onto the stack

Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 2

Popped item: 9

Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 3

Top item: 8

Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 4

Size of stack: 2

Stack Operations:

1. Push
2. Pop
3. Peek
4. Size
5. Exit

Enter your choice: 5

**Queue:** class Queue: def \_\_init\_\_(self): self.items = []

def is\_empty(self): return self.items == []

def enqueue(self, item):

self.items.append(item)

def dequeue(self):

if not self.is\_empty():

return self.items.pop(0) else:

return "Queue is empty"

def peek(self):

if not self.is\_empty():

return self.items[0] else:

return "Queue is empty"

def size(self):

return len(self.items)

if \_\_name\_\_ == "\_\_main\_\_": q = Queue()

q.enqueue(1)

q.enqueue(2)

q.enqueue(3)

print("Queue size:", q.size()) print("Front of the queue:", q.peek()) print("Dequeue:", q.dequeue()) print("Dequeue:", q.dequeue()) print("Dequeue:", q.dequeue()) print("Dequeue:", q.dequeue()) **Output of Queue:**

Queue size: 3

Front of the queue: 1

Dequeue: 1

Dequeue: 2

Dequeue: 3

Dequeue: Queue is empty

**Linked List:** class Node:

def \_\_init\_\_(self, data): self.data = data self.next = None class LinkedList:

def \_\_init\_\_(self):

self.head = None def append(self, data): new\_node = Node(data)

if not self.head: self.head = new\_node

return current = self.head while current.next:

current = current.next current.next = new\_node def insert(self, position, data):

new\_node = Node(data) if position == 0: new\_node.next = self.head self.head = new\_node

return current = self.head for \_ in range(position - 1):

if current is None:

raise IndexError("Position out of range") current = current.next new\_node.next = current.next

current.next = new\_node def remove(self, data):

if self.head is None: raise Exception("List is empty") if self.head.data == data: self.head = self.head.next return current = self.head while current.next:

if current.next.data == data: current.next = current.next.next return current = current.next raise ValueError("Data not found in the list") def replace(self, old\_data, new\_data):

current = self.head while current:

if current.data == old\_data: current.data = new\_data return current = current.next raise ValueError("Data not found in the list") def search\_position(self, data):

position = 0 current = self.head while current:

if current.data == data:

return position current = current.next position += 1 raise ValueError("Data not found in the list") def size(self):

count = 0

current = self.head while current:

count += 1

current = current.next return count def traverse(self): elements = [] current = self.head while current:

elements.append(current.data) current = current.next return elements llist = LinkedList()

while True:

print("\n1. Append") print("2. Insert") print("3. Remove") print("4. Replace") print("5. Search Position") print("6. Size") print("7. Display") print("8. Exit")

choice = int(input("Enter your choice: "))

if choice == 1: data = int(input("Enter data to append: ")) llist.append(data) elif choice == 2:

position = int(input("Enter position to insert: ")) data = int(input("Enter data to insert: ")) llist.insert(position, data)

elif choice == 3: data = int(input("Enter data to remove: ")) llist.remove(data) elif choice == 4:

old\_data = int(input("Enter data to replace: ")) new\_data = int(input("Enter new data: ")) llist.replace(old\_data, new\_data) elif choice == 5:

data = int(input("Enter data to search position: "))

try:

position = llist.search\_position(data) print(f"Position of {data}: {position}") except ValueError as e:

print(e) elif choice == 6:

print("Size of linked list:", llist.size()) elif choice == 7:

print("Linked list:", llist.traverse()) elif choice == 8:

print("Exiting...") break

else:

print("Invalid choice. Please try again.") **Output of Linked List:**

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 1

Enter data to append: 45

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 1

Enter data to append: 89

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 7

Linked list: [45, 89]

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 2

Enter position to insert: 1

Enter data to insert: 99

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 7

Linked list: [45, 99, 89]

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 5

Enter data to search position: 89

Position of 89: 2

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 4

Enter data to replace: 45

Enter new data: 100

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 7

Linked list: [100, 99, 89]

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 6 Size of linked list: 3

1. Append
2. Insert
3. Remove
4. Replace
5. Search Position
6. Size
7. Display
8. Exit

Enter your choice: 8

Exiting…

**Conclusion:** In Python, stacks, queues, and linked lists are fundamental data structures serving different purposes. Stacks follow last-in, first-out (LIFO) order, useful for managing function calls or undo mechanisms. Queues adhere to first-in, first-out (FIFO) order, ideal for task scheduling or breadth-first search algorithms. Linked lists offer dynamic memory allocation and efficient insertion/deletion, making them suitable for scenarios requiring flexibility in data storage and manipulation. In Python, these structures can be implemented using lists with built-in methods, or custom classes for linked lists. Understanding their characteristics and implementations is essential for developing efficient algorithms and applications.